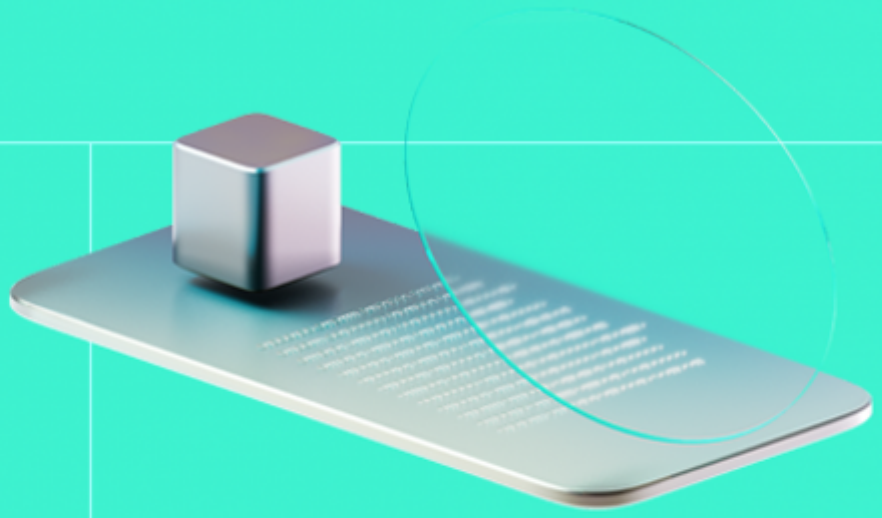
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Zharta

Date: 16/10/2024

We express our gratitude to the Zharta team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Zharta is a peer-to-pool lending platform that allows users to lend and loan out NFTs.

Document

Name	Smart Contract Code Review and Security Analysis Report for Zharta
Audited By	Seher Saylik, Giovanni Franchi
Approved By	Ataberk Yavuzer
Website	https://www.zharta.io/
Changelog	24/09/2024 - Preliminary Report
	16/10/2024 - Final Report
Platform	Ethereum
Language	Vyper
Tags	P2P Lending, NFT Lending, ERC721
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/Zharta/lending-protocol-v2
Commit	1e32d64

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

6	4	1	1
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	1
High	1
Medium	1
Low	3

Vulnerability	Severity
F-2024-6069 - Lender Paying Borrower Broker Fee in settle_loan Enables Loan Theft	Critical
F-2024-6139 - Delegation Not Revoked After Loan Closure	High
F-2024-6063 - Potential Exploit via Lender Blacklisting in the USDC Contract Leading to Collateral Seizure	Medium
F-2024-6065 - Owner Can Frontrun Borrowers by Setting High Protocol Fees	Low
F-2024-6068 - Use of tx.origin Leads to Phishing Attack Risks When setApprovalForAll is Granted	Low
F-2024-6156 - No Cap on Settlement Fees	Low

Documentation quality

- Functional requirements are provided.
- Technical description is provided.
- NatSpecs are detailed.

Code quality

- The code is well written and is clear in its intentions.
- Some gas inefficiencies are present.
- The code follows best practises in terms of readability and input validation.
- The development environment is configured.

Test coverage

Code coverage of the project is **98.00%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage are mostly covered.
- Some fuzz testing is present.

Table of Contents

System Overview

Zharta is a Peer-to-peer ERC-721 backed lending protocol with the following contract:

P2PLendingNfts — It is the core contract. It enables users to provide and take loan offers backed by ERC-721. This contract facilitates the creation and acceptance of loan offers, where lenders provide loans and borrowers use NFTs as collateral. The contract supports both ERC-721 and CryptoPunks NFTs, offering flexibility in collateral types. Additionally, it integrates with DelegateRegistry v2 for delegation functionality, allowing borrowers to delegate their collateral.

Two Interest Models: Lenders can choose between two types of interest structures—fixed or pro-rata. Fixed interest applies uniformly across the loan duration, while pro-rata interest is calculated based on the time the loan is held.

P2PLendingControl — It is an auxiliary module that manages state variables related to supported collateral and trait types. It allows the owner to update and modify these parameters as needed.

Privileged roles

- The owner of the contract can set protocol fees.
- The owner of the contract can change the protocol wallet.
- The owner of the contract can enable or disable an authorised proxy.
- The owner of the contract can propose a new owner.
- The owner of the contract can enable or disable supported ERC-721 collections.
- The owner of the contract can set the `type_root` for a specific collateral.

Potential Risks

- A potential risk arises from using `tx.origin` to identify the borrower's address when a loan is initiated through an authorised proxy. If this feature remains part of the protocol, as highlighted in issue **F-2024-6068**, it could be vulnerable to phishing attacks, particularly if the authorized proxy fails to implement robust access control measures. Since the current audit does not cover the code of authorised proxies, this security risk cannot be fully addressed or verified within the scope of this assessment. It is therefore recommended to only enable authorised proxies whose code has been audited.
- Delegation of collateral is handled through an **external registry**. If the delegation registry fails or is compromised, it could lead to improper management of collateral or delegation rights.
- Compilers used for converting smart contract code (e.g., Solidity or Vyper) into executable bytecode can contain hidden bugs, posing a risk to the integrity of the final deployed contract. Even after a thorough audit of the source code, undetected compiler bugs may introduce unintended behaviour or vulnerabilities that cannot be identified during typical code reviews.
- The loan duration in the signed offer can potentially be set to zero or very low durations. If this occurs, borrowers could immediately be at risk of default upon loan creation, as the loan would mature instantly. This could result in borrowers losing their collateralized assets without an opportunity to fulfill repayment obligations. It is the responsibility of the users to carefully review all signed offer data to ensure fairness and accuracy.
- The protocol utilises Merkle trees to allow lenders to specify particular traits of collateral they are willing to lend against. For computational efficiency, the `trait_root` of a specific collateral address **is calculated off-chain**, making it impossible to verify its correctness within the current audit. If a trait type is incorrectly mapped to a token ID during the construction of the `trait_root`, there is a potential scenario where a borrower could secure a loan intended for a specific trait type that the collateral does not actually possess.
- When **pro-rata** is set to **True** at loan creation, and if the loan is replaced with a higher interest rate by lenders, borrowers may lose their broker fees for the remainder of the loan duration.

Findings

Vulnerability Details

F-2024-6069 - Lender Paying Borrower Broker Fee in settle_loan Enables Loan Theft - Critical

Description: In the `settle_loan()` function, the lender ends up paying the settlement fees, including the `borrower_broker_settlement_fee_bps`. This is not the intended behaviour, as the borrower should be responsible for their broker's fees. This issue arises due to the calculation and deduction of settlement fees from the total amount to be sent to the lender.

```
@external
def settle_loan(loan: Loan):

    """
    @notice Settle a loan.
    @param loan The loan to be settled.
    """

    assert self._is_loan_valid(loan), "invalid loan"
    assert block.timestamp <= loan.maturity, "loan defaulted"
    assert self._check_user(loan.borrower), "not borrower"

    interest: uint256 = self._compute_settlement_interest(loan)
    settlement_fees_total: uint256 = 0
    settlement_fees: DynArray[FeeAmount, MAX_FEES] = []
    settlement_fees, settlement_fees_total = self._get_settlement_fees(loan,
interest)

    self.loans[loan.id] = empty(bytes32)

    self._receive_funds(loan.borrower, loan.amount + interest)

    self._send_funds(loan.lender, loan.amount + interest - settlement_fees_to
tal)
    for fee in settlement_fees:
        self._send_funds(fee.wallet, fee.amount)

    self._transfer_collateral(loan.borrower, loan.collateral_contract, loan.c
ollateral_token_id)

    log LoanPaid(
```

```
            loan.id,
            loan.borrower,
            loan.lender,
            loan.payment_token,
            loan.amount,
            interest,
            settlement_fees
        )
```

As showed by this specific section of the code:

```
        self._receive_funds(loan.borrower, loan.amount + interest)

        self._send_funds(loan.lender, loan.amount + interest - settlement_fees_to
tal)
```

The borrower is required to repay the loan amount plus interest. However, all settlement fees are deducted from the amount the lender should receive, effectively making the lender bear the cost of the brokerage service, which should typically be the borrower's responsibility, as the name implies.

This creates an exploit where the borrower can manipulate the settlement fee mechanism by setting their own address as the `borrower_broker`. By doing so, the borrower can claim a percentage of the settlement fees—potentially even exceeding `100%` of the interest owed. This enables the borrower to siphon off nearly all remaining funds after the deduction of other fees, leaving the lender with almost nothing.

Assets:

- P2PLendingNfts.vy [https://github.com/Zharta/lending-protocol-v2/tree/feat/initial-impl]

Status: Fixed

Classification

Impact: 5/5

Likelihood: 5/5

Exploitability: Independent

Complexity: Medium

Severity: Critical

Recommendations

Remediation: It is recommended to adjust the calculation to ensure that the borrower pays their broker's settlement fees. This can be done by modifying the `_receive_funds_from_caller()` function call to include the borrower's broker settlement fee.

Resolution: **Remediation: (revised commit: 65ec5b6):** Now the `create_loan()`, `replace_loan()` and `replace_loan_lender()` functions implement a correct fee accountancy in which is the borrower the one responsible for paying the `borrower_broker_fee`. This fix therefore prevents a borrower to make usage of the `borrower_broker_fee` to perform malicious accountancy manipulations against the lender.

Evidences

POC

Reproduce:

Steps:

- Set up an extreme borrower broker fee with almost all the loan value going to the borrower acting as their own broker.
- Create an offer.
- Sign the offer using the lender's private key.
- Mint and approve the NFT collateral for the borrower.
- Deposit the principal amount (minus fees) by the lender into the protocol.
- Create the loan on-chain with the signed offer and broker fees.
- Create a loan object for off-chain validation.
- Verify that the loan hash matches the on-chain stored hash.
- Approve the loan settlement amount in USDC by the borrower.
- Settle the loan through the `p2p_nfts_usdc` contract.
- Capture initial and final balances of both the borrower and lender.
- Assert that the lender receives close to zero, while the borrower gains almost all the funds through broker fees.

```
def test_borrower_exploits_broker_fee_to_leave_lender_with_almost_zero(p2p_nf
ts_usdc, usdc, borrower, lender, lender_key, bayc, now, protocol_fees, borrow
er_broker_fee):
    # Step 1: Set the parameters for an extreme borrower broker fee
    extreme_borrower_broker_fee_bps = 99999  # Almost everything after other
fees
    token_id = 1  # Token ID for the collateral

    # Step 2: Create a new offer
```

```
    extreme_offer = Offer(
        principal=1000,
        interest=100,
        payment_token=usdc.address,
        duration=100,
        origination_fee_amount=0,
        broker_upfront_fee_amount=0,
        broker_settlement_fee_bps=extreme_borrower_broker_fee_bps,
        broker_address=borrower,  # Borrower is their own broker
        collateral_contract=bayc.address,
        collateral_min_token_id=token_id,
        collateral_max_token_id=token_id,
        expiration=now + 100,
        lender=lender,
        pro_rata=False,
        size=1
    )

    # Step 3: Sign the offer with the lender's key
    extreme_signed_offer = sig
```

See more

Results:

initial_borrower_balance= `1000000000000`
initial_lender_balance= `1000000001000`
extreme_offer.principal= `1000`
final_borrower_balance - initial_borrower_balance= `883`
final_lender_balance - initial_lender_balance= `-912`

Files:

```
De268E')])], pro_rata=False)
initial_borrower_balance=1000000000000
initial_lender_balance=1000000001000
extreme_offer.principal=1000
final_borrower_balance - initial_borrower_balance=883
final_lender_balance - initial_lender_balance=-912
PASSED
```

F-2024-6139 - Delegation Not Revoked After Loan Closure - High

Description:

In the `create_loan()` function, a delegation is created for the address passed by the borrower. This means that if there are other contracts where the delegation of a specific token ID of a collection is relevant for claiming rewards, accessing awards, etc., the borrower can still act as a delegated address even if the token ID now belongs to another address that has used it for collateral again in the contract. This issue originates from the fact that the `DelegateRegistry` doesn't override delegated addresses when a delegation is written in it.

```
@external
def create_loan(
    offer: SignedOffer,
    collateral_token_id: uint256,
    delegate: address,
    borrower_broker_upfront_fee_amount: uint256,
    borrower_broker_settlement_fee_bps: uint256,
    borrower_broker: address
) -> bytes32:

    """
    @notice Create a loan.
    @param offer The signed offer.
    @param collateral_token_id The ID of the collateral token.
    @param delegate The address of the delegate. If empty, no delegation is s
et.
    @param borrower_broker_upfront_fee_amount The upfront fee amount for the
borrower broker.
    @param borrower_broker_settlement_fee_bps The settlement fee basis points
relative to the interest for the borrower broker.
    @param borrower_broker The address of the borrower broker.
    @return The ID of the created loan.
    """



    assert self._is_offer_signed_by_lender(offer, offer.offer.lender), "offer
not signed by lender"
    assert offer.offer.expiration > block.timestamp, "offer expired"
    assert offer.offer.payment_token == payment_token, "invalid payment token
"
    assert offer.offer.origination_fee_amount <= offer.offer.principal, "orig
ination fee gt principal"

    assert self.whitelisted[offer.offer.collateral_contract], "collateral not
whitelisted"
```

```
    assert offer.offer.collateral_min_token_id <= collateral_token_id, "token
id below offer range"
    assert offer.offer.collateral_max_token_id >= collateral_token_id, "token
id above offer range"

    fees: DynArray[Fee, MAX_FEES] = self._get_loan_fees(offer.offer, borrower
_broker_upfront_fee_amount, borrower_broker_settlement_fee_bps, borrower_brok
er)
    total_upfront_fees: uint256 = 0
    for fee in fees:
        total_upfront_fees += fee.upfront_amount

    loan: Loan = Loan({
        id: empty(bytes32),
        amount: offer.offer.principal,
        interest: offer.offer.interest,
        payment_token: offer.offer.payment_token,
        maturity: block.timestamp + offer.offer.duration,
        start_time: block.timestamp,
        borrower: msg.sender if not self.authorized_proxies[msg.sender] else
tx.origin,
        lender: offer.offer.lender,
        collateral_contract: offer.offer.collateral_contract,
        collateral_token_id: collateral_token_id,
        fees: fees,
        pro_rata: offer.offer.pro_rata
    })
    loan.id = self._compute_loan_id(loan)

    assert self.loans[loan.id] == empty(bytes32), "loan already exists"
    self._check_and_update_offer_state(offer)
    self.loans[loan.id] = self._loan_state_hash(loan)

    self._store_collateral(loan.borrower, loan.collateral_contract, loan.coll
ateral_token_id)
    self._transfer_funds(loan.lender, loan.borrower, loan.amount - total_upfr
ont_fees + offer.offer.broker_upfront_fee_amount)

    for fee in fees:
        if fee.type != FeeType.ORIGINATION_FEE and fee.upfront_amount > 0:
            self._transfer_funds(loan.lender, fee.wallet, fee.upfront_amount)

    self._set_delegation(delegate, loan.collateral_contract, loan.collateral_
token_id, delegate != empty(address))

    log LoanCreated(
        loan.id,
        loan.amount,
        loan.interest,
```

```
        loan.payment_token,
        loan.maturity,
        loan.start_time,
        loan.borrower,
        loan.lender,
        loan.collateral_contract,
        loan.collateral_token_id,
        loan.fees,
        loan.pro_rata,
        self._compute_signed_offer_id(offer)
    )
    return loan.id
```

```
@external
def settle_loan(loan: Loan):

    """
    @notice Settle a loan.
    @param loan The loan to be settled.
    """

    assert self._is_loan_valid(loan), "invalid loan"
    assert block.timestamp <= loan.maturity, "loan defaulted"
    assert self._check_user(loan.borrower), "not borrower"

    interest: uint256 = self._compute_settlement_interest(loan)
    settlement_fees_total: uint256 = 0
    settlement_fees: DynArray[FeeAmount, MAX_FEES] = []
    settlement_fees, settlement_fees_total = self._get_settlement_fees(loan,
interest)

    self.loans[loan.id] = empty(bytes32)

    self._receive_funds(loan.borrower, loan.amount + interest)

    self._send_funds(loan.lender, loan.amount + interest - settlement_fees_to
tal)
    for fee in settlement_fees:
        self._send_funds(fee.wallet, fee.amount)

    self._transfer_collateral(loan.borrower, loan.collateral_contract, loan.c
ollateral_token_id)

    log LoanPaid(
        loan.id,
        loan.borrower,
        loan.lender,
        loan.payment_token,
```

```
        loan.amount,
        interest,
        settlement_fees
    )
```

```
@external
def claim_defaulted_loan_collateral(loan: Loan):

    """
    @notice Claim defaulted loan collateral.
    @param loan The loan whose collateral is to be claimed. The loan maturity
must have been passed.
    """

    assert self._is_loan_valid(loan), "invalid loan"
    assert block.timestamp > loan.maturity, "loan not defaulted"
    assert self._check_user(loan.lender), "not lender"

    self.loans[loan.id] = empty(bytes32)

    self._transfer_collateral(loan.lender, loan.collateral_contract, loan.col
lateral_token_id)

    log LoanCollateralClaimed(
        loan.id,
        loan.borrower,
        loan.lender,
        loan.collateral_contract,
        loan.collateral_token_id
    )
```

The delegation remains active after the loan is settled or defaulted. This behaviour is not intended as the functional requirements state that:

> The protocol integrates with delegation.xyz delegation registry V2, potentially allowing for NFT utility during the loan period. Delegation is set when a new loan is created and remains in place until the loan is settled, regardless of whether the collateral is claimed or returned to the borrower. Delegation is set in full, not using the registry's subdelegation feature.

If the delegation is not revoked when the loan is closed (either by being settled or defaulted), the borrower can still act as the owner of the collateral. This can lead to the borrower claiming rewards or benefits that should belong to the new owner, causing potential financial and security issues.

The following issue arises in the `delegateContract()` where it can be observed that the slot location for storing the delegation depends on the hash of the parameters passed to the function.

```
hash = Hashes.contractHash(msg.sender, rights, to, contract_);
bytes32 location = Hashes.location(hash);
address loadedFrom = _loadFrom(location);
```

Example:

- Bob takes a loan putting `ExampleNft` as collateral and passing his address as a delegated address.
- Bob defaults the loan that he was owing to `Alice`, now Alice is the owner of `ExampleNft`.
- Alice now uses `ExampleNft` as collateral for taking a loan and passes her address as a delegated address.
- Throughout the duration of Alice's loan, Bob can act as a delegated address, potentially claiming rewards that should rightfully belong to Alice.

Assets:

- P2PLendingNfts.vy [https://github.com/Zharta/lending-protocol-v2/tree/feat/initial-impl]

Status: Fixed

Classification

Impact: 3/5

Likelihood: 5/5

Exploitability: Independent

Complexity: Simple

Severity: High

Recommendations

Remediation: Revoke the delegation when the loan is closed (either by being settled or defaulted) by adding the necessary logic in the `settle_loan()` and `claim_defaulted_loan_collateral()` functions.

Resolution: **Remediation: (revised commit: 80b76a0):** The protocol is now correctly removing, if it exists, a delegation to a specific collateral

when a loan is either repaid or defaulted. The `Loan` struct has been modified with the addition of the `delegate` address.

Evidences

POC

Reproduce:

Although not involving directly the in-scope contract, this basic POC demonstrates how setting a delegator address for a specific token id and subsequently setting a new delegator does not override the previous delegation.

Steps:

- `P2PLendingNFT` delegates Alice
- It is asserted that Alice is delegated by `P2PLendingNFT`
- `P2PLendingNFT` delegates John
- It is asserted that John is delegated by `P2PLendingNFT`
- It is also asserted that Alice is still delegated by `P2PLendingNFT`
- Eventually the only way to disable a delegation is to call the previous function passing as a `enable` parameter false

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.24;


import {Test, console} from "forge-std/Test.sol";
import {IDelegateRegistry} from "../src/interfaces/IDelegationRegistry.sol";


contract TestDelegation is Test {
    address public constant DELEGATION = 0x00000000000000447e69651d841bD8D104
Bed493;
    address public constant AZUKI = 0xED5AF388653567Af2F388E6224dC7C4b3241C54
4;
    uint256 public constant AZUKI_ID = 5556;
    address public P2PLendingNfts = makeAddr("P2PLendingNfts");
    address public alice = makeAddr("alice");
    address public john = makeAddr("john");


    function setUp() public {
        uint256 forkId = vm.createFork(vm.envString("RPC_URL"));
        vm.selectFork(forkId);
    }
```

```
    function test_delegation() public {
        vm.startPrank(P2PLendingNfts);
        IDelegateRegistry(DELEGATION).delegateERC721(alice, AZUKI, AZUKI_ID,
bytes32(0), true);
        bool isAliceDelgated = IDelegateRegistry(DELEGATION).checkDelegateFor
ERC721(alice, P2PLendingNfts, AZUKI, AZUKI_ID, bytes32(0));
        assert(isAliceDelgated);
        IDelegateRegistry(DELEGATION).delegateERC721(john, AZUKI, AZUKI_ID, b
ytes32(0), true);
        bool isJohnDelgated = IDelegateRegistry(
```

See more

Files:

```
[⠊] Solc 0.8.24 finished in 3.51s
Compiler run successful!

Ran 1 test for test/TestDelegation.sol:TestDelegation
[PASS] test_delegation() (gas: 332855)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 6.50s (4.28s CPU time)

Ran 1 test suite in 6.53s (6.50s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

F-2024-6063 - Potential Exploit via Lender Blacklisting in the USDC Contract Leading to Collateral Seizure - Medium

Description:

In the current implementation of the `settle_loan()` function, if the lender is blacklisted in the payment token contract (such as USDC as functional requirements suggest), the loan settlement will fail during the fund transfer to the lender. The function attempts to send the loan repayment (principal + interest) to the lender using the `_send_funds()` function. However, if the lender is blacklisted in the payment token contract, the transfer will revert, resulting in the borrower retaining the collateral without completing the loan repayment.

```
@external
def settle_loan(loan: Loan):

    """
    @notice Settle a loan.
    @param loan The loan to be settled.
    """

    assert self._is_loan_valid(loan), "invalid loan"
    assert block.timestamp <= loan.maturity, "loan defaulted"
    assert self._check_user(loan.borrower), "not borrower"

    interest: uint256 = self._compute_settlement_interest(loan)
    settlement_fees_total: uint256 = 0
    settlement_fees: DynArray[FeeAmount, MAX_FEES] = []
    settlement_fees, settlement_fees_total = self._get_settlement_fees(loan,
interest)

    self.loans[loan.id] = empty(bytes32)

    self._receive_funds(loan.borrower, loan.amount + interest)

    self._send_funds(loan.lender, loan.amount + interest - settlement_fees_to
tal)
    for fee in settlement_fees:
        self._send_funds(fee.wallet, fee.amount)

    self._transfer_collateral(loan.borrower, loan.collateral_contract, loan.c
ollateral_token_id)

    log LoanPaid(
        loan.id,
        loan.borrower,
```

```
        loan.lender,
        loan.payment_token,
        loan.amount,
        interest,
        settlement_fees
    )
```

```
@internal
def _send_funds(_to: address, _amount: uint256):
    assert IERC20(payment_token).transfer(_to, _amount), "error sending funds
"
```

As the USDC contract on mainnet shows, if an address is blacklisted the transaction will revert

```
    function transfer(address to, uint256 value)
        external
        override
        whenNotPaused
        notBlacklisted(msg.sender)
        notBlacklisted(to)
        returns (bool)
    {
        _transfer(msg.sender, to, value);
        return true;
    }
```

Assets:

- P2PLendingNfts.vy [https://github.com/Zharta/lending-protocol-v2/tree/feat/initial-impl]

Status: Fixed

Classification

Impact: 4/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: It is recommended a `pull` over `push` pattern in which the correct execution of the `settle_loan()` function does not depend on the successful transfer of funds to the lender.

Resolution: The given issue is resolved only when ERC20 tokens that return `true` upon transfer are used. This is because the response in the `_send_funds()` function depends on the `transfer()` function returning `true`. If the `transfer()` function does not return a value, the transaction will still succeed, but the response will be an empty byte(It will be converted to “False” in `_send_funds()`). As a result, the `_send_funds()` function would both transfer funds to the loan lender and update the `pending_transfer` by the amount transferred. To prevent double spending, it is critical to ensure that the USDC contract used returns `true` upon transfer.

(Revised commit: **de753d8**)

F-2024-6065 - Owner Can Frontrun Borrowers by Setting High Protocol Fees - Low

Description:

The `set_protocol_fee()` function allows the owner to set the protocol fees without any upper limit or timelock. This can lead to a potential frontrunning issue where the owner can set very high protocol fees just before a borrower calls the `create_loan()` function. This effectively reduces the amount the borrower receives and the amount the lender will get back, leading to unfair terms for both the borrower and the lender.

```
@external
def set_protocol_fee(protocol_upfront_fee: uint256, protocol_settlement_fee:
uint256):

    """
    @notice Set the protocol fee
    @dev Sets the protocol fee to the given value and logs the event. Admin f
unction.
    @param protocol_upfront_fee The new protocol upfront fee.
    @param protocol_settlement_fee The new protocol settlement fee.
    """

    assert msg.sender == self.owner, "not owner"

    log ProtocolFeeSet(self.protocol_upfront_fee, self.protocol_settlement_fe
e, protocol_upfront_fee, protocol_settlement_fee)
    self.protocol_upfront_fee = protocol_upfront_fee
    self.protocol_settlement_fee = protocol_settlement_fee
```

As the function shows, no upper bounds nor time-locks are present to protect users against owner's possible misbehaviour when the following function is executed.

```
@external
def create_loan(
    offer: SignedOffer,
    collateral_token_id: uint256,
    delegate: address,
    borrower_broker_upfront_fee_amount: uint256,
    borrower_broker_settlement_fee_bps: uint256,
    borrower_broker: address
) -> bytes32:

    """
    @notice Create a loan.
```

```
    @param offer The signed offer.
    @param collateral_token_id The ID of the collateral token.
    @param delegate The address of the delegate. If empty, no delegation is s
et.
    @param borrower_broker_upfront_fee_amount The upfront fee amount for the
borrower broker.
    @param borrower_broker_settlement_fee_bps The settlement fee basis points
relative to the interest for the borrower broker.
    @param borrower_broker The address of the borrower broker.
    @return The ID of the created loan.
    """



    assert self._is_offer_signed_by_lender(offer, offer.offer.lender), "offer
not signed by lender"
    assert offer.offer.expiration > block.timestamp, "offer expired"
    assert offer.offer.payment_token == payment_token, "invalid payment token
"
    assert offer.offer.origination_fee_amount <= offer.offer.principal, "orig
ination fee gt principal"

    assert self.whitelisted[offer.offer.collateral_contract], "collateral not
whitelisted"
    assert offer.offer.collateral_min_token_id <= collateral_token_id, "token
id below offer range"
    assert offer.offer.collateral_max_token_id >= collateral_token_id, "token
id above offer range"

    fees: DynArray[Fee, MAX_FEES] = self._get_loan_fees(offer.offer, borrower
_broker_upfront_fee_amount, borrower_broker_settlement_fee_bps, borrower_brok
er)
    total_upfront_fees: uint256 = 0
    for fee in fees:
        total_upfront_fees += fee.upfront_amount

    loan: Loan = Loan({
        id: empty(bytes32),
        amount: offer.offer.principal,
        interest: offer.offer.interest,
        payment_token: offer.offer.payment_token,
        maturity: block.timestamp + offer.offer.duration,
        start_time: block.timestamp,
        borrower: msg.sender if not self.authorized_proxies[msg.sender] else
tx.origin,
        lender: offer.offer.lender,
        collateral_contract: offer.offer.collateral_contract,
        collateral_token_id: collateral_token_id,
        fees: fees,
        pro_rata: offer.offer.pro_rata
```

```
    })
    loan.id = self._compute_loan_id(loan)

    assert self.loans[loan.id] == empty(bytes32), "loan already exists"
    self._check_and_update_offer_state(offer)
    self.loans[loan.id] = self._loan_state_hash(loan)

    self._store_collateral(loan.borrower, loan.collateral_contract, loan.coll
ateral_token_id)
    self._transfer_funds(loan.lender, loan.borrower, loan.amount - total_upfr
ont_fees + offer.offer.broker_upfront_fee_amount)

    for fee in fees:
        if fee.type != FeeType.ORIGINATION_FEE and fee.upfront_amount > 0:
            self._transfer_funds(loan.lender, fee.wallet, fee.upfront_amount)

    self._set_delegation(delegate, loan.collateral_contract, loan.collateral_
token_id, delegate != empty(address))

    log LoanCreated(
        loan.id,
        loan.amount,
        loan.interest,
        loan.payment_token,
        loan.maturity,
        loan.start_time,
        loan.borrower,
        loan.lender,
        loan.collateral_contract,
        loan.collateral_token_id,
        loan.fees,
        loan.pro_rata,
        self._compute_signed_offer_id(offer)
    )
    return loan.id
```

Potential exploit scenario:

1. The owner sets very high protocol fees using the `set_protocol_fee` function.
2. A borrower calls the `create_loan` function to create a new loan.
3. The high protocol fees reduce the amount the borrower receives and the amount the lender will get back.
4. The borrower and the lender end up with unfair loan terms due to the sudden increase in protocol fees.

Assets:

- P2PLendingNfts.vy [https://github.com/Zharta/lending-protocol-v2/tree/feat/initial-impl]

Status: Accepted

Classification

Impact: 5/5

Likelihood: 2/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to implement an upper bound value to cap fees to a reasonable amount and also to provide a time-lock system to let users be able to act according to the hypothetical change of these crucial variables.

Resolution: **Remediation: (revised commit: 57a1285):** Despite the protocol now enforces upper bounds for protocol fees, the present finding could not be considered as fixed due to the fact that a time-lock mechanism has not been enforced therefore leading to users being frontrunned still being possible. The client is aware of the current situation and accepts the risk.

F-2024-6068 - Use of tx.origin Leads to Phishing Attack Risks When setApprovalForAll is Granted - Low

Description:

The `create_loan()` function in the contract utilises the `tx.origin` to assign the `borrower` when the transaction originates from one of the approved proxy contracts. This creates a vulnerability where an exploiter can take advantage of users who have granted `setApprovalForAll` to the protocol. By phishing the user into interacting with a malicious contract, the exploiter can initiate a loan with terms that significantly benefit the exploiter and potentially lead to the loss of the user's NFT collateral.

```
@external
def create_loan(
    offer: SignedOffer,
    collateral_token_id: uint256,
    delegate: address,
    borrower_broker_upfront_fee_amount: uint256,
    borrower_broker_settlement_fee_bps: uint256,
    borrower_broker: address
) -> bytes32:

    """
    @notice Create a loan.
    @param offer The signed offer.
    @param collateral_token_id The ID of the collateral token.
    @param delegate The address of the delegate. If empty, no delegation is s
et.
    @param borrower_broker_upfront_fee_amount The upfront fee amount for the
borrower broker.
    @param borrower_broker_settlement_fee_bps The settlement fee basis points
relative to the interest for the borrower broker.
    @param borrower_broker The address of the borrower broker.
    @return The ID of the created loan.
    """



    assert self._is_offer_signed_by_lender(offer, offer.offer.lender), "offer
not signed by lender"
    assert offer.offer.expiration > block.timestamp, "offer expired"
    assert offer.offer.payment_token == payment_token, "invalid payment token
"
    assert offer.offer.origination_fee_amount <= offer.offer.principal, "orig
ination fee gt principal"

    assert self.whitelisted[offer.offer.collateral_contract], "collateral not
```

```
whitelisted"
    self._validate_token_ids(offer.offer, collateral_token_id)

    fees: DynArray[Fee, MAX_FEES] = self._get_loan_fees(offer.offer, borrower
_broker_upfront_fee_amount, borrower_broker_settlement_fee_bps, borrower_brok
er)
    total_upfront_fees: uint256 = 0
    for fee in fees:
        total_upfront_fees += fee.upfront_amount

    loan: Loan = Loan({
        id: empty(bytes32),
        amount: offer.offer.principal,
        interest: offer.offer.interest,
        payment_token: offer.offer.payment_token,
        maturity: block.timestamp + offer.offer.duration,
        start_time: block.timestamp,
        borrower: msg.sender if not self.authorized_proxies[msg.sender] else
tx.origin,
        lender: offer.offer.lender,
        collateral_contract: offer.offer.collateral_contract,
        collateral_token_id: collateral_token_id,
        fees: fees,
        pro_rata: offer.offer.pro_rata
    })
    loan.id = self._compute_loan_id(loan)

    assert self.loans[loan.id] == empty(bytes32), "loan already exists"
    self._check_and_update_offer_state(offer)
    self.loans[loan.id] = self._loan_state_hash(loan)

    self._store_collateral(loan.borrower, loan.collateral_contract, loan.coll
ateral_token_id)
    self._transfer_funds(loan.lender, loan.borrower, loan.amount - total_upfr
ont_fees + offer.offer.broker_upfront_fee_amount)

    for fee in fees:
        if fee.type != FeeType.ORIGINATION_FEE and fee.upfront_amount > 0:
            self._transfer_funds(loan.lender, fee.wallet, fee.upfront_amount)

    self._set_delegation(delegate, loan.collateral_contract, loan.collateral_
token_id, delegate != empty(address))

    log LoanCreated(
        loan.id,
        loan.amount,
        loan.interest,
        loan.payment_token,
        loan.maturity,
```

```
            loan.start_time,
            loan.borrower,
            loan.lender,
            loan.collateral_contract,
            loan.collateral_token_id,
            loan.fees,
            loan.pro_rata,
            self._compute_signed_offer_id(offer)
        )
        return loan.id
```

Attack Scenario:

1. **Granting Approval:** The victim grants `setApprovalForAll` to the protocol for his NFTs, allowing the protocol to manage their NFTs.
2. **Phishing:** The exploiter tricks the user into interacting with a malicious contract, which then interacts with one of the protocol's approved proxy contracts.
3. **Loan Creation Abuse:** The malicious contract creates a loan via the `create_loan()` function with an `offer.origination_fee_amount` almost equal to the `offer.offer.principal`. This ensures that the actual loan amount provided by the lender is minimal, covering only the origination and protocol fees.
4. **NFT Loss or Repayment:** The user (victim) is either forced to repay the loan or lose their NFT as collateral. Additionally, the exploiter can manipulate the loan's duration to be extremely short, further increasing the risk of the victim losing the NFT quickly.

Security Impact considerations:

- **Phishing Risk:** An attacker can deceive the user into interacting with a malicious contract that exploits the proxy mechanism to create unfavourable loans.
- **Collateral Loss:** If the loan has extremely high origination fees or very short terms, the victim can lose their NFT, effectively allowing the attacker to steal it.
- **Minimal Cost to Attacker:** The attacker only needs to cover the origination fee and protocol fee, resulting in a minimal cost for the loan.

It's important to note that part of the attack surface related to the hypothetical exploit involves the use of `authorized_proxies`, which are outside the scope of the current audit. However, the repository under review provides an example of an `authorized_proxy` in the `P2PNftsProxy.vy` contract.

```
@external
def create_loan(
    offer: SignedOffer,
    collateral_token_id: uint256,
    delegate: address,
    borrower_broker_upfront_fee_amount: uint256,
    borrower_broker_settlement_fee_bps: uint256,
    borrower_broker: address
) -> bytes32:
    return P2PLendingNfts(self.p2p_lending_nfts).create_loan(
        offer,
        collateral_token_id,
        delegate,
        borrower_broker_upfront_fee_amount,
        borrower_broker_settlement_fee_bps,
        borrower_broker
    )
```

As shown in the example above, there is no verification to ensure that the address originating the transaction is the actual borrower when interacting with an authorized proxy. This omission opens up a potential attack vector where an attacker could exploit this proxy mechanism.

Although this issue is only partially within the scope of this audit, we believe it is helpful to highlight this possible vulnerability to the team, as it could be exploited if not properly addressed.

Assets:

- P2PLendingNfts.vy [https://github.com/Zharta/lending-protocol-v2/tree/feat/initial-impl]

Status: Mitigated

Classification

Impact: 5/5

Likelihood: 2/5

Exploitability: Semi-Dependent

Complexity: Complex

Severity: Low

Recommendations

Remediation: Two possible solutions are recommended:

- A simple and robust solution is to remove the ability for approved proxy contracts to initiate loans on behalf of users. This would ensure that `msg.sender` is always the borrower, and the transaction is directly authorised by the user.
- If removing the proxy logic is not feasible, the proxy contracts must be carefully audited and secured. Proxy contracts should validate that the actual caller (`msg.sender`) is the intended borrower, ensuring that `msg.sender == tx.origin`. This would prevent attackers from using malicious intermediaries to initiate loan transactions on behalf of unsuspecting users.

Resolution: The team acknowledges the risks associated with using `tx.origin` in this context. However, the contract flow in question is deemed essential for the ecosystem's future integrations. As a result, the team is fully committed to auditing every contract that will be integrated with `P2PLendingNfts.vy`. In the current audit, the identified scenarios could not be verified, and therefore, this issue should be considered mitigated.

F-2024-6156 - No Cap on Settlement Fees - Low

Description:

In the contract, there is no upper limit on the settlement fees that can be charged upon loan settlement. This issue arises from the way the settlement fees are calculated using the `interest_bps` (basis points) for each fee in the `Loan` struct. Because these fees are computed as a percentage of the settlement interest, there is a potential scenario where the cumulative settlement fees could exceed the total loan principal and interest.

If the settlement fees surpass the amount owed by the borrower (i.e., the principal plus the accrued interest), the loan settlement process could fail, preventing borrowers from successfully closing their loans. This can lead to both a significant financial burden for borrowers and operational issues within the protocol, as the contract might be left in an unusable state with unresolved loans.

Affected function:

```
@external
def settle_loan(loan: Loan):

    """
    @notice Settle a loan.
    @param loan The loan to be settled.
    """

    assert self._is_loan_valid(loan), "invalid loan"
    assert block.timestamp <= loan.maturity, "loan defaulted"
    assert self._check_user(loan.borrower), "not borrower"

    interest: uint256 = self._compute_settlement_interest(loan)
    settlement_fees_total: uint256 = 0
    settlement_fees: DynArray[FeeAmount, MAX_FEES] = []
    settlement_fees, settlement_fees_total = self._get_settlement_fees(loan,
interest)

    self.loans[loan.id] = empty(bytes32)

    self._receive_funds(loan.borrower, loan.amount + interest)

    self._send_funds(loan.lender, loan.amount + interest - settlement_fees_to
tal)
    for fee in settlement_fees:
        self._send_funds(fee.wallet, fee.amount)

    self._transfer_collateral(loan.borrower, loan.collateral_contract, loan.c
```

```
ollateral_token_id)

    log LoanPaid(
        loan.id,
        loan.borrower,
        loan.lender,
        loan.payment_token,
        loan.amount,
        interest,
        settlement_fees
    )
```

Assets:

- P2PLendingNfts.vy [https://github.com/Zharta/lending-protocol-v2/tree/feat/initial-impl]

Status: Fixed

Classification

Impact: 3/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Implement a check that ensures settlement fees do not exceed the total repayment amount, which includes the loan principal and interest. This prevents potential problems with insufficient funds or locked balances.

Resolution: The Zharta team introduced a new validation in `_get_loan_fees()` function to ensure that the combined percentage of the protocol settlement fee and the lender broker settlement fee does not exceed 100%. This check prevents potential underflows in the `settle_loan()` function when the settlement fees exceed the sum of the loan principal and interest.

(Revised commit: **3a613a8**)

Observation Details

F-2024-6163 - Missing Zero Address Validation - Info

Description:

In the `__init__` function, several parameters passed to the function are addresses of external contracts or essential components for the protocol operation. Currently, only the `_protocol_wallet` and `_payment_token` parameters undergo zero address check, ensuring it is not set to the empty address (`0x0`). However, the other address parameters (`_delegation_registry`,, `_cryptopunks`,) do not have similar zero address checks.

If any of the external contract addresses (such as `_delegation_registry`, etc.) are set to the zero address, interactions with these contracts could fail.

```
@external
def __init__(
    _payment_token: address,
    _delegation_registry: address,
    _cryptopunks: address,
    _protocol_upfront_fee: uint256,
    _protocol_settlement_fee: uint256,
    _protocol_wallet: address
):

    """
    @notice Initialize the contract with the given parameters.
    @param _payment_token The address of the payment token.
    @param _delegation_registry The address of the delegation registry.
    @param _cryptopunks The address of the CryptoPunksMarket contract.
    @param _protocol_upfront_fee The percentage (bps) of the principal paid t
o the protocol at origination.
    @param _protocol_settlement_fee The percentage (bps) of the interest paid
to the protocol at settlement.
    @param _protocol_wallet The address where the protocol fees are accrued.
    """

    assert _protocol_wallet != empty(address), "wallet is the zero address"
    assert _payment_token != empty(address), "payment token is zero"

    self.owner = msg.sender
    payment_token = _payment_token
    delegation_registry = DelegationRegistry(_delegation_registry)
    cryptopunks = CryptoPunksMarket(_cryptopunks)
    self.protocol_upfront_fee = _protocol_upfront_fee
```

```
    self.protocol_settlement_fee = _protocol_settlement_fee
    self.protocol_wallet = _protocol_wallet

    offer_sig_domain_separator = keccak256(
        _abi_encode(
            DOMAIN_TYPE_HASH,
            keccak256(ZHARTA_DOMAIN_NAME),
            keccak256(ZHARTA_DOMAIN_VERSION),
            chain.id,
            self
        )
    )
```

Assets:

- P2PLendingNfts.vy [https://github.com/Zharta/lending-protocol-v2/tree/feat/initial-impl]

Status: Fixed

Recommendations

Remediation: Add zero address checks for the following parameters in the `__init__` function:

1. `_delegation_registry`
2. `_cryptopunks`

Resolution: The Zharta team introduced the necessary zero address checks. (Revised commit: **57a1285**)

F-2024-6213 - Immutable Variables Can Be Constants - Info

Description:

The following immutable variables in the `P2PLendingNfts.vy` file can be declared as constants since the `cryptopunks` contract is only deployed on the Ethereum mainnet. Declaring them as constants can save gas.

```
payment_token: public(immutable(address))

delegation_registry: public(immutable(DelegationRegistry))

cryptopunks: public(immutable(CryptoPunksMarket))
```

Constants are never padded in the contract bytecode, whereas immutables are reserved a full 32-byte word in the contract bytecode, even if their value requires fewer bytes (for example, an immutable defined as a uint32 type) will still be padded to 32 bytes).

Assets:

- P2PLendingNfts.vy [https://github.com/Zharta/lending-protocol-v2/tree/feat/initial-impl]

Status: Fixed

Recommendations

Remediation: It is recommended to mark these variables as constants and assign them a predefined value before deployment.

Resolution: The client highlighted that this contract will be deployed on multiple chains therefore it is necessary to leave the aforementioned variables settable at the deploy time. The client also added that for `cryptopunks`, which are not present in other chains, the value will be set as `address(0)`.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project’s performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/Zharta/lending-protocol-v2
Commit	1e32d64ad042701e8157c7f93a6a0da7e3779c7c
Remediation Commit	1e7048e248349fab4d720cba43b47f64e0735dbc
Whitepaper	N/A
Requirements	https://github.com/Zharta/lending-protocol-v2/blob/feat/initial-impl/README.md
Technical Requirements	https://github.com/Zharta/lending-protocol-v2/blob/feat/initial-impl/README.md

Asset	Type
P2PLendingControl.vy [https://github.com/Zharta/lending-protocol-v2/tree/feat/initial-impl]	Smart Contract
P2PLendingNfts.vy [https://github.com/Zharta/lending-protocol-v2/tree/feat/initial-impl]	Smart Contract

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.